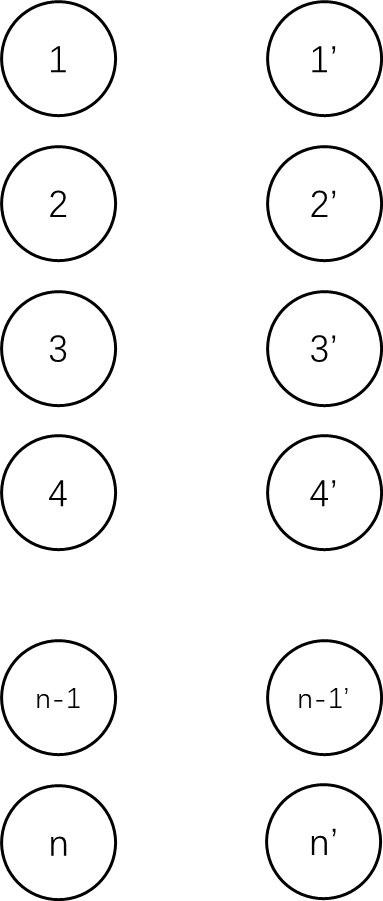
1. 关于|S|<=|N(S)|与存在完美匹配互为充要条件的直观理解：



反证法（仅供理解）：

假设当前有n对节点，并且不存在完美匹配。

如果1号节点没有被匹配，则我们找到了集合S，满足|S|>|N(S)|，所以1号节点需要被匹配，假设匹配到了1‘。

那么1‘节点在这之前一定也被匹配过了，不然我们就可以将1和1‘匹配起来，得到一个更好的解。所以假设1’与2匹配，则我们又找到了一个集合S={1,2}，满足|S|>|N(S)|，所以2号节点也已经匹配了，假设与2’匹配。

这样一直找下去，假设前n-1个节点都被匹配上了，则n-1’要与n匹配，所以左边一共有n个节点匹配，而右边只有n-1个节点，所以这与匹配定义矛盾。

1. 最大流模板使用
2. N表示图中所有可能的顶点数，即包括可能存在的超级源点和超级汇点；
3. M表示图中所有可能的边数，因此需要考虑构图之后的边数和所有的反向边；
4. #define后的括号不能去掉，同时init()函数需要每求一次最大流之前运行一次；
5. 如果原图中定义了反向边，直接加；
6. 如果原图中定义了平行边，也直接加；
7. 求最大流的s和t需要自己设置，一般来说，如果节点编号从0开始，则源点设为n，汇点设为n+1。节点从1开始，源点设为0，汇点设为n+1。
8. Print(v)函数是一个用来debug的函数，用来输出节点v之前（不包括v），所有点的连接情况。
9. Upstream(s,n)是推流函数，用来输出除去源点s之外，源点可达的点数。

/\*

\* Dinic algo for max flow

\*

\* This implementation assumes that #nodes, #edges, and capacity on each edge <= INT\_MAX,

\* which means INT\_MAX is the best approximation of INF on edge capacity.

\* The total amount of max flow computed can be up to LLONG\_MAX (not defined in this file),

\* but each 'dfs' call in 'dinic' can return <= INT\_MAX flow value.

\*/

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <string.h>

#include <assert.h>

#include <queue>

#include <vector>

#include <algorithm>

#define N (100+2)

#define M (N\*N+4\*N)

typedef long long LL;

using namespace std;

struct edge {

int v, cap, next;

};

edge e[M];

int head[N], level[N], cur[N];

int num\_of\_edges;

/\*

\* When there are multiple test sets, you need to re-initialize before each

\*/

void dinic\_init(void) {

num\_of\_edges = 0;

memset(head, -1, sizeof(head));

return;

}

int add\_edge(int u, int v, int c1, int c2) {

int& i=num\_of\_edges;

assert(c1>=0 && c2>=0 && c1+c2>=0); // check for possibility of overflow

e[i].v = v;

e[i].cap = c1;

e[i].next = head[u];

head[u] = i++;

e[i].v = u;

e[i].cap = c2;

e[i].next = head[v];

head[v] = i++;

return i;

}

void print\_graph(int n) {

for (int u=0; u<n; u++) {

printf("%d: ", u);

for (int i=head[u]; i>=0; i=e[i].next) {

printf("%d(%d)", e[i].v, e[i].cap);

}

printf("\n");

}

return;

}

/\*

\* Find all augmentation paths in the current level graph

\* This is the recursive version

\*/

int dfs(int u, int t, int bn) {

if (u == t) return bn;

int left = bn;

for (int &i=cur[u]; i>=0; i=e[i].next) {

int v = e[i].v;

int c = e[i].cap;

if (c > 0 && level[u]+1 == level[v]) {

int flow = dfs(v, t, min(left, c));

if (flow > 0) {

e[i].cap -= flow;

e[i^1].cap += flow;

cur[u] = i;

left -= flow;

if (!left) break;

}

}

}

if (left > 0) level[u] = 0;

return bn - left;

}

bool bfs(int s, int t) {

memset(level, 0, sizeof(level));

level[s] = 1;

queue<int> q;

q.push(s);

while (!q.empty()) {

int u = q.front();

q.pop();

if (u == t) return true;

for (int i=head[u]; i>=0; i=e[i].next) {

int v = e[i].v;

if (!level[v] && e[i].cap > 0) {

level[v] = level[u]+1;

q.push(v);

}

}

}

return false;

}

LL dinic(int s, int t) {

LL max\_flow = 0;

while (bfs(s, t)) {

memcpy(cur, head, sizeof(head));

max\_flow += dfs(s, t, INT\_MAX);

}

return max\_flow;

}

int upstream(int s, int n) {

int cnt = 0;

vector<bool> visited(n);

queue<int> q;

visited[s] = true;

q.push(s);

while (!q.empty()) {

int u = q.front();

q.pop();

for (int i=head[u]; i>=0; i=e[i].next) {

int v = e[i].v;

if (e[i].cap > 0 && !visited[v]) {

visited[v] = true;

q.push(v);

cnt++;

}

}

}

return cnt; // excluding s

}

int main() {

int m, n, s, t;

int pig[M+1], pre[M+1];

bool con[N+1];

FILE \*fin;

/\*fin = fopen("pigs.dat", "r");

assert(fin);\*/

fin = stdin;

fscanf(fin, "%d %d", &m, &n);

dinic\_init();

s = 0, t = n+1;

for (int i=1; i<=m; i++) {

fscanf(fin, "%d", &pig[i]);

}

memset(pre, -1, sizeof(pre));

for (int i=1; i<=n; i++) {

int nkeys;

fscanf(fin, "%d", &nkeys);

memset(con, 0, sizeof(con));

int cap = 0;

for (int j=1; j<=nkeys; j++) {

int ph;

fscanf(fin, "%d", &ph);

int cust = pre[ph];

if (cust < 0) {

cap += pig[ph];

} else if (!con[cust]) {

add\_edge(cust, i, INT\_MAX, 0);

}

pre[ph] = i;

}

if (cap > 0) {

add\_edge(s, i, cap, 0);

}

int npigs;

fscanf(fin, "%d", &npigs);

if (npigs > 0) {

add\_edge(i, t, npigs, 0);

}

}

//print\_graph(n+2);

int flow = dinic(s, t);

printf("%d\n", flow);

return 0;

}

1. 最大匹配使用了贪心+增广的匈牙利算法

/\*

\* hungarian algo for maximum\_matching

\*/

#include <cstdio>

#include <iostream>

#include <vector>

using namespace std;

vector<vector<int> > lnklst;

vector<int> l, r;

vector<bool> visited;

/\* make all vectors one element bigger in case the index starts from 1 instead of 0 \*/

void init(int n1, int n2) {

lnklst.clear(); lnklst.resize(n1+1);

l.clear(); l.resize(n1+1,-1);

r.clear(); r.resize(n2+1,-1);

return;

}

void add\_edge(int u, int v) {

lnklst[u].push\_back(v);

return;

}

bool dfs(int u) {

for (int i=0; i<lnklst[u].size(); i++) {

int v = lnklst[u][i];

if (visited[v]) continue;

visited[v] = true;

if (r[v] < 0 || dfs(r[v])) {

l[u] = v;

r[v] = u;

return true;

}

}

return false;

}

int greedy\_match(int n1) {

int match = 0;

for (int u=0; u<n1; u++) {

if (l[u] < 0) {

for (int i=0; i<lnklst[u].size(); i++) {

int v = lnklst[u][i];

if (r[v] < 0) {

l[u] = v;

r[v] = u;

match++;

break;

}

}

}

}

return match;

}

int hungarian(void) {

int n1 = l.size();

int n2 = r.size();

int match = greedy\_match(n1);

for (int u=0; u<n1; u++) {

if (l[u] < 0) {

visited.clear();

visited.resize(n2);

if (dfs(u)) {

match++;

}

}

}

return match;

}

int main() {

//freopen("path\_cover.dat", "r", stdin);

int n, m;

while (cin >> m >> n) {

init(n, n);

for (int i=0; i<m; i++) {

int u, v;

cin >> u >> v;

add\_edge(u, v);

}

int max\_match = hungarian();

cout << n-max\_match << endl;

}

return 0;

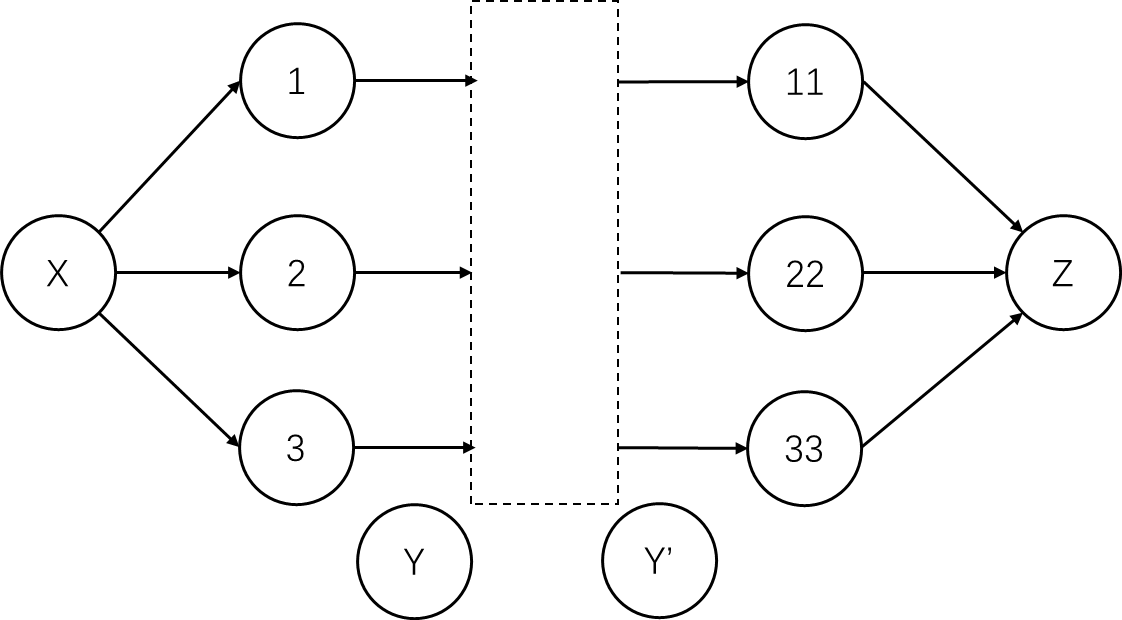
}

1. 不交路径的传递性

假设点X到Y有k条不交路径，Y到Z有k条不交路径，则是否能推出X到Z也有k条不交路径？

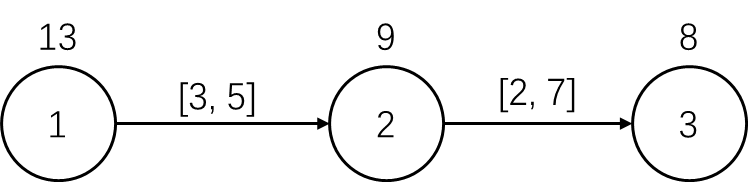
结论：可以

证明：反证法，假设X到Z只有k-1条不交路径。根据Menger’s定理，我们至少需要移除k-1条边F，使得X到Z不连通，则如果Y在F的左侧，则说明移除了k-1条边后，Y到Z不连通，与Y到Z有k条不交路径矛盾。反之，如果Y在F的右侧，则与X到Y有k条不交路径矛盾。

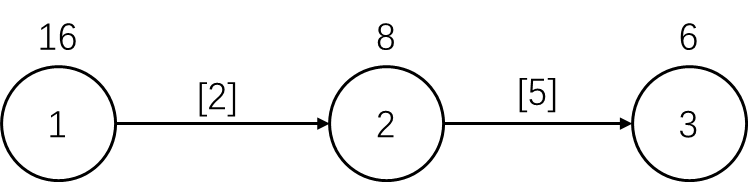


1. 拥有上下界网络流的转化

原图：



新图：



先将每个顶点加上出边的流量，减去入边的流量，再将每条边流量上下界相减，作为新的容量即可。